inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #21
Caches I

CPS today!

2005-11-14

There is one handout today at the front and back of the room!

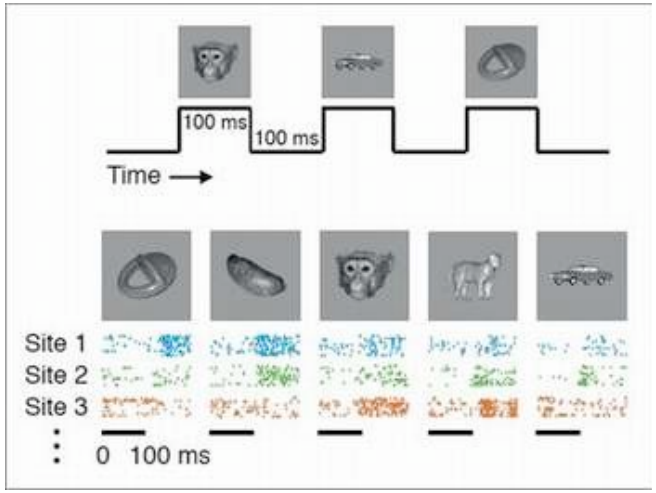
Lecturer PSOE, new dad Dan Garcia

www.cs.berkeley.edu/~ddgarcia

The matrix…reality? ⇒

MIT neuroscientists can now decipher part of the code involved in recognizing visual objects! A "classifier" was used on the monkey brain signals.

www.physorg.com/news7879.html



Review

- Pipeline challenge is hazards
  - Forwarding helps w/many data hazards
  - Delayed branch helps with control hazard in 5 stage pipeline
- More aggressive performance:
  - Superscalar
  - Out-of-order execution
- You can be creative with your pipelines
- Learn from our top 10 worst SW bugs…
  - Test, test, test. Expect the unexpected.
  - Design w/failure as possibility! Redundancy!

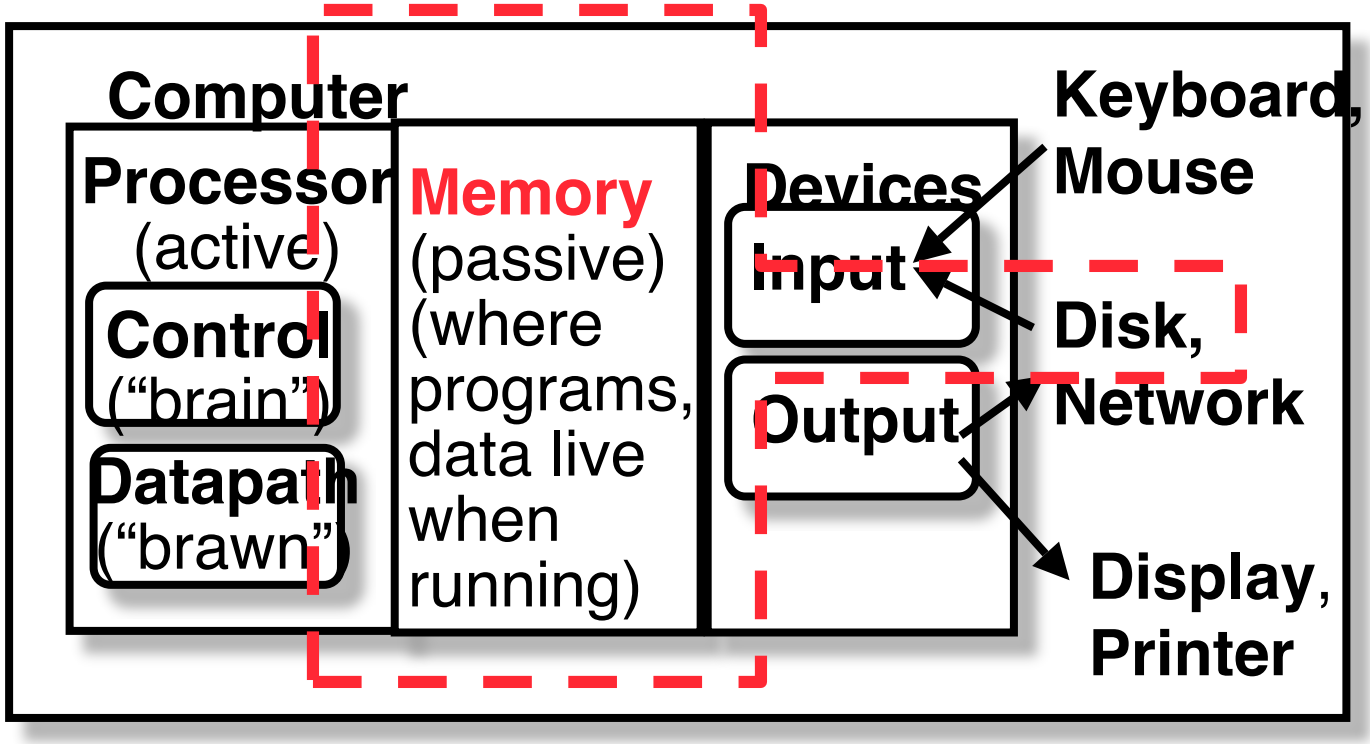
Big Ideas so far

- 15 weeks to learn big ideas in CS&E
  - Principle of abstraction, used to build systems as layers
  - Pliable Data: a program determines what it is
  - Stored program concept: instructions just data
  - Compilation v. interpretation to move down layers of system
  - Greater performance by exploiting parallelism (pipeline)
  - Principle of Locality, exploited via a memory hierarchy (cache)
  - Principles/Pitfalls of Performance Measurement

Where are we now in 61C?

- **Architecture!** (aka "Systems")
  - CPU Organization
    - Datapath
    - Control
  - Pipelining
  - **Caches**
  - **Virtual Memory**
  - **I / O**
  - **Networks**
  - **Performance**

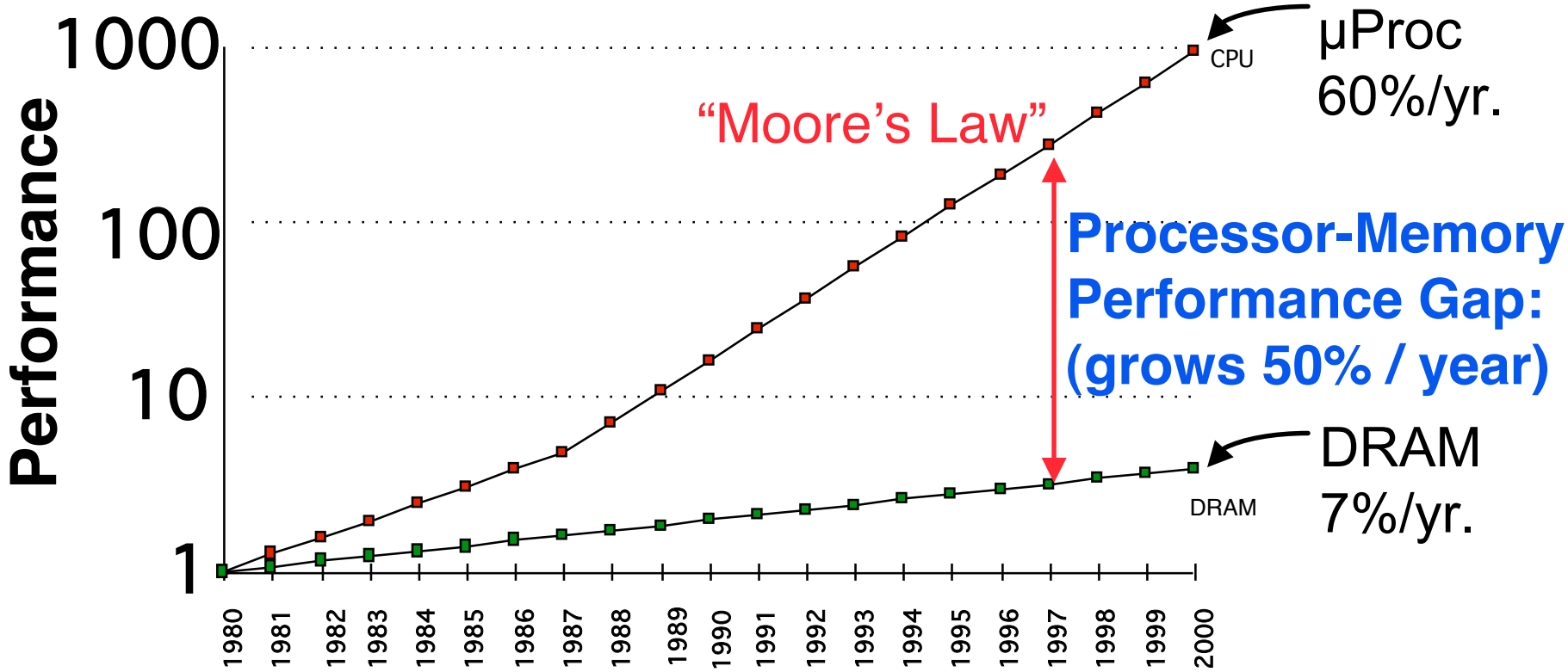
The Big Picture



Memory Hierarchy (1/3)

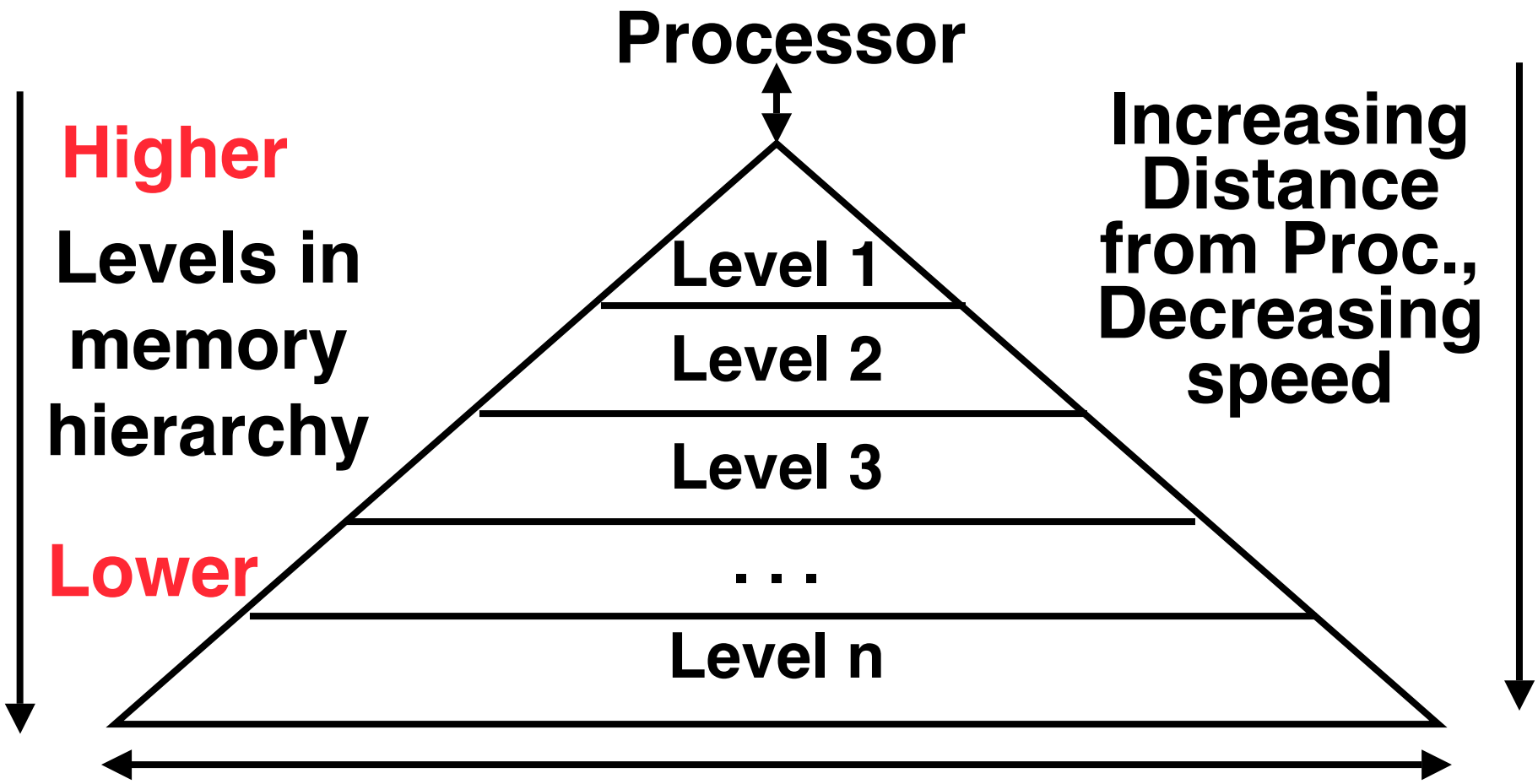
- Processor
  - executes instructions on order of nanoseconds to picoseconds
  - holds a small amount of code and data in registers
- Memory
  - More capacity than registers, still limited
  - Access time ~50-100 ns
- Disk
  - HUGE capacity (virtually limitless)
  - VERY slow: runs ~milliseconds

Review: Why We Use Caches



- 1989 first Intel CPU with cache on chip
- 1998 Pentium III has two levels of cache on chip

Memory Hierarchy (2/3)



As we move to deeper levels the latency goes up and price per bit goes down.

Q: Can $/bit go up as move deeper?

Memory Hierarchy (3/3)

- If level closer to Processor, it must be:
  - smaller
  - faster
  - subset of lower levels (contains most recently used data)
- Lowest Level (usually disk) contains all available data
- Other levels?

Memory Caching

- We've discussed three levels in the hierarchy: processor, memory, disk
- Mismatch between processor and memory speeds leads us to add a new level: a memory cache
- Implemented with SRAM technology: faster but more expensive than DRAM memory.
  - "S" = Static, no need to refresh, ~10ns
  - "D" = Dynamic, need to refresh, ~60ns
  - `arstechnica.com/paedia/r/ram_guide/ram_guide.part1-1.html`

Memory Hierarchy Analogy: Library (1/2)

- You're writing a term paper (Processor) at a table in Doe
- Doe Library is equivalent to disk
  - essentially limitless capacity
  - very slow to retrieve a book
- Table is memory
  - smaller capacity: means you must return book when table fills up
  - easier and faster to find a book there once you've already retrieved it

Memory Hierarchy Analogy: Library (2/2)

- Open books on table are cache
  - smaller capacity: can have very few open books fit on table; again, when table fills up, you must close a book
  - much, much faster to retrieve data
- Illusion created: whole library open on the tabletop
  - Keep as many recently used books open on table as possible since likely to use again
  - Also keep as many books on table as possible, since faster than going to library

Memory Hierarchy Basis

- Disk contains everything.
- When Processor needs something, bring it into to all higher levels of memory.
- Cache contains copies of data in memory that are being used.
- Memory contains copies of data on disk that are being used.
- Entire idea is based on Temporal Locality: if we use it now, we'll want to use it again soon (a Big Idea)

Cache Design

- How do we organize cache?
- Where does each memory address map to?

  (Remember that cache is subset of memory, so multiple memory addresses map to the same cache location.)
- How do we know which elements are in cache?
- How do we quickly locate them?

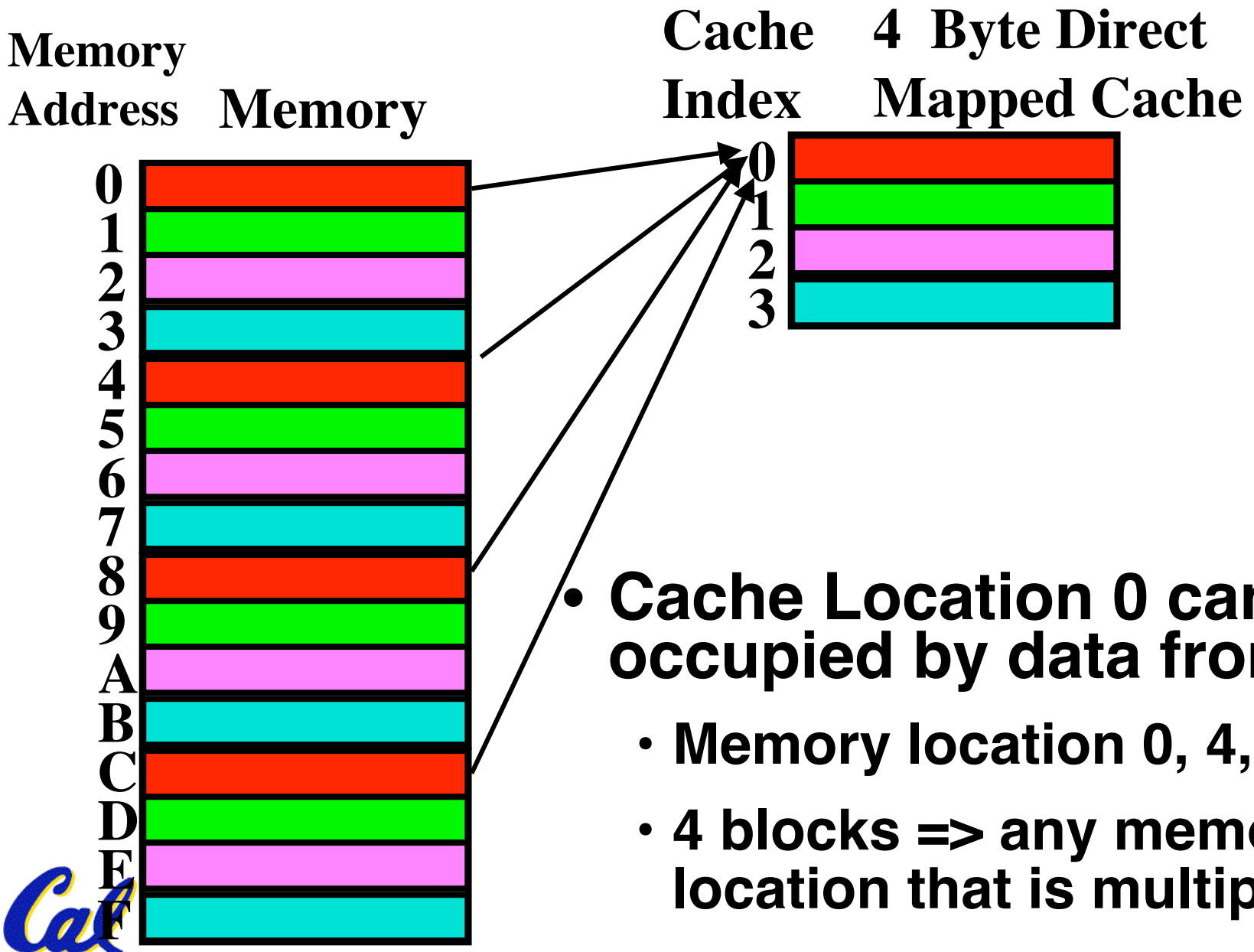
Administrivia

- Dan's Wed's OH this week moved a few hours earlier to 10-11am
- Project 3 due Friday

Direct-Mapped Cache (1/2)

- In a direct-mapped cache, each memory address is associated with one possible block within the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - Block is the unit of transfer between cache and memory

Direct-Mapped Cache (2/2)



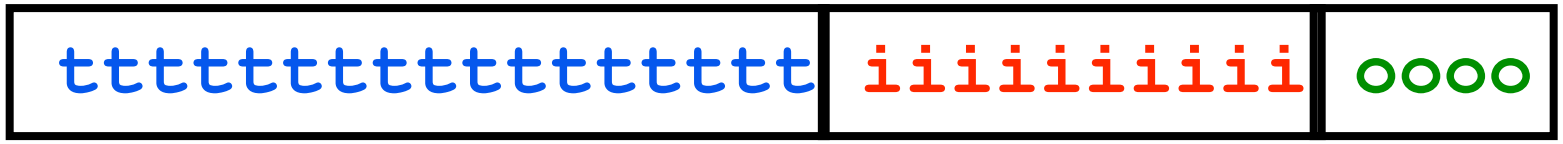
- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - 4 blocks => any memory location that is multiple of 4

Issues with Direct-Mapped

Tag	Index	Offset

- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Answer: divide memory address into three fields

HEIGHT WIDTH



tag to check if have correct block

index to select block

byte offset within block

Direct-Mapped Cache Terminology

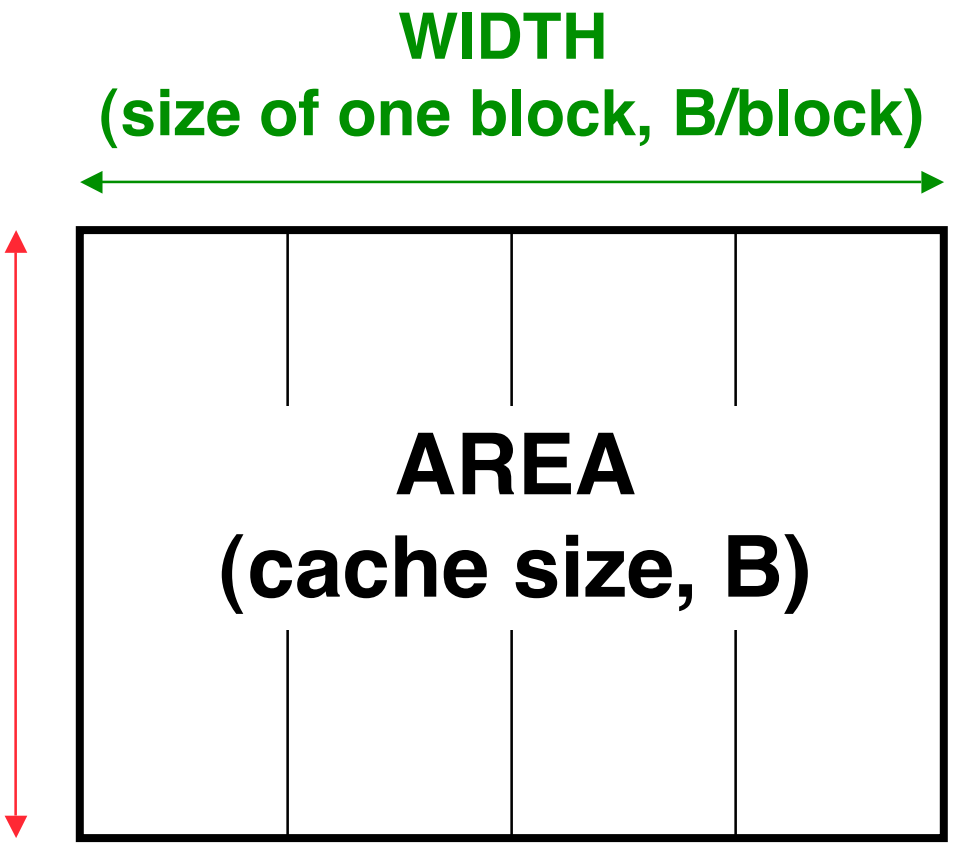
- All fields are read as unsigned integers.
- Index: specifies the cache index (which "row" of the cache we should look in)
- Offset: once we've found correct block, specifies which byte within the block we want -- I.e., which "column"
- Tag: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location

TIO Dan's great cache mnemonic

AREA (cache size, B)
= HEIGHT (# of blocks)
*** WIDTH (size of one block, B/block)**

$$2^{(H+W)} = 2^H * 2^W$$

Tag	Index	Offset



Direct-Mapped Cache Example (1/3)

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture
- Offset
  - need to specify correct byte within a block
  - block contains 4 words

    = 16 bytes

    = 2^4 bytes
  - need 4 bits to specify correct byte

Direct-Mapped Cache Example (2/3)

- Index: (~index into an "array of blocks")
  - need to specify correct row in cache
  - cache contains 16 KB = 2^{14} bytes
  - block contains 2^4 bytes (4 words)
  - # blocks/cache

$$= \frac{\text{bytes/cache}}{\text{bytes/block}}$$

$$= \frac{2^{14}\ \text{bytes/cache}}{2^4\ \text{bytes/block}}$$

$$= 2^{10}\ \text{blocks/cache}$$

  - need 10 bits to specify this many rows

Direct-Mapped Cache Example (3/3)

- Tag: use remaining bits as tag
  - tag length = addr length - offset - index
    = 32 - 4 - 10 bits
    = 18 bits
  - so tag is leftmost 18 bits of memory address
- Why not full 32 bit address as tag?
  - All bytes within block need same address (4b)
  - Index must be same for every address within a block, so its redundant in tag check, thus can leave off to save memory (10 bits in this example)

Caching Terminology

- When we try to read memory, 3 things can happen:

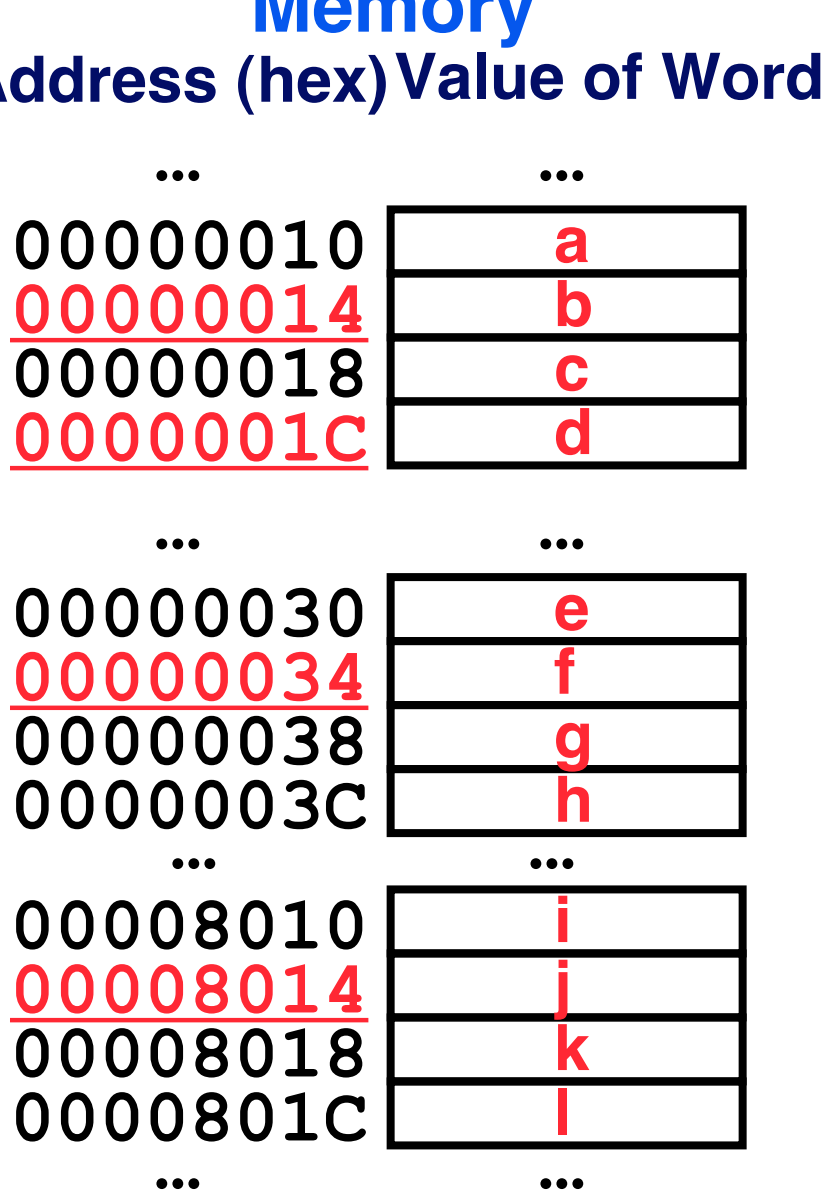
1. cache hit:
   cache block is valid and contains proper address, so read desired word

2. cache miss:
   nothing in cache in appropriate block, so fetch from memory

3. cache miss, block replacement:
   wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

Accessing data in a direct mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks
- Read 4 addresses
  1. 0x00000014
  2. 0x0000001C
  3. 0x00000034
  4. 0x00008014
- Memory values on right:
  - only cache/ memory level of hierarchy



Accessing data in a direct mapped cache

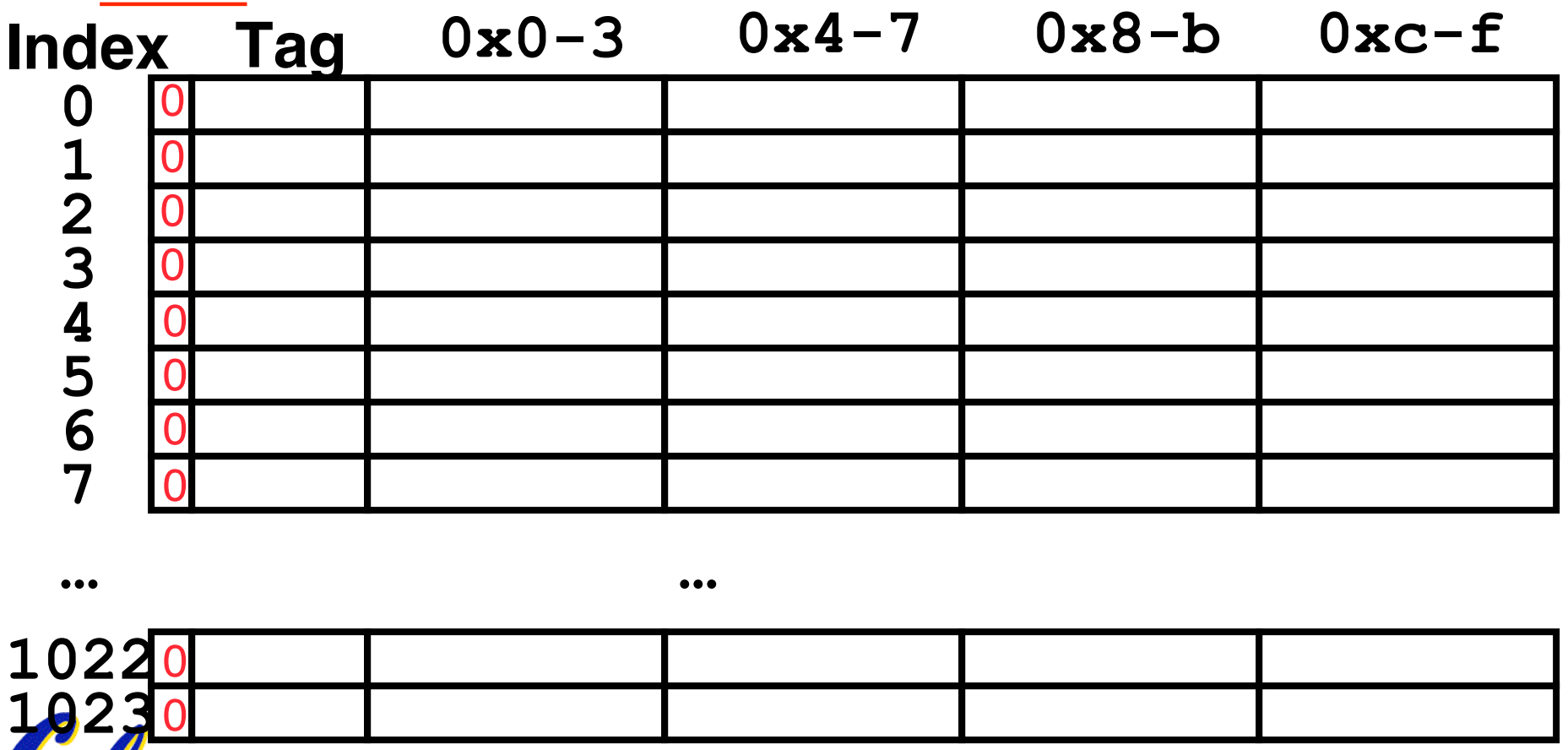
- 4 Addresses:
  - `0x00000014, 0x0000001C, 0x00000034, 0x00008014`
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

```
00000000000000000000 0000000001 0100
00000000000000000000 0000000001 1100
00000000000000000000 0000000011 0100
00000000000000000010 0000000001 0100
```

Tag Index Offset

16 KB Direct Mapped Cache, 16B blocks

- **Valid bit:** determines whether anything is stored in that row (when computer initially turned on, all entries invalid)



1. Read 0x00000014

- 00000000000000000000 0000000001 0100

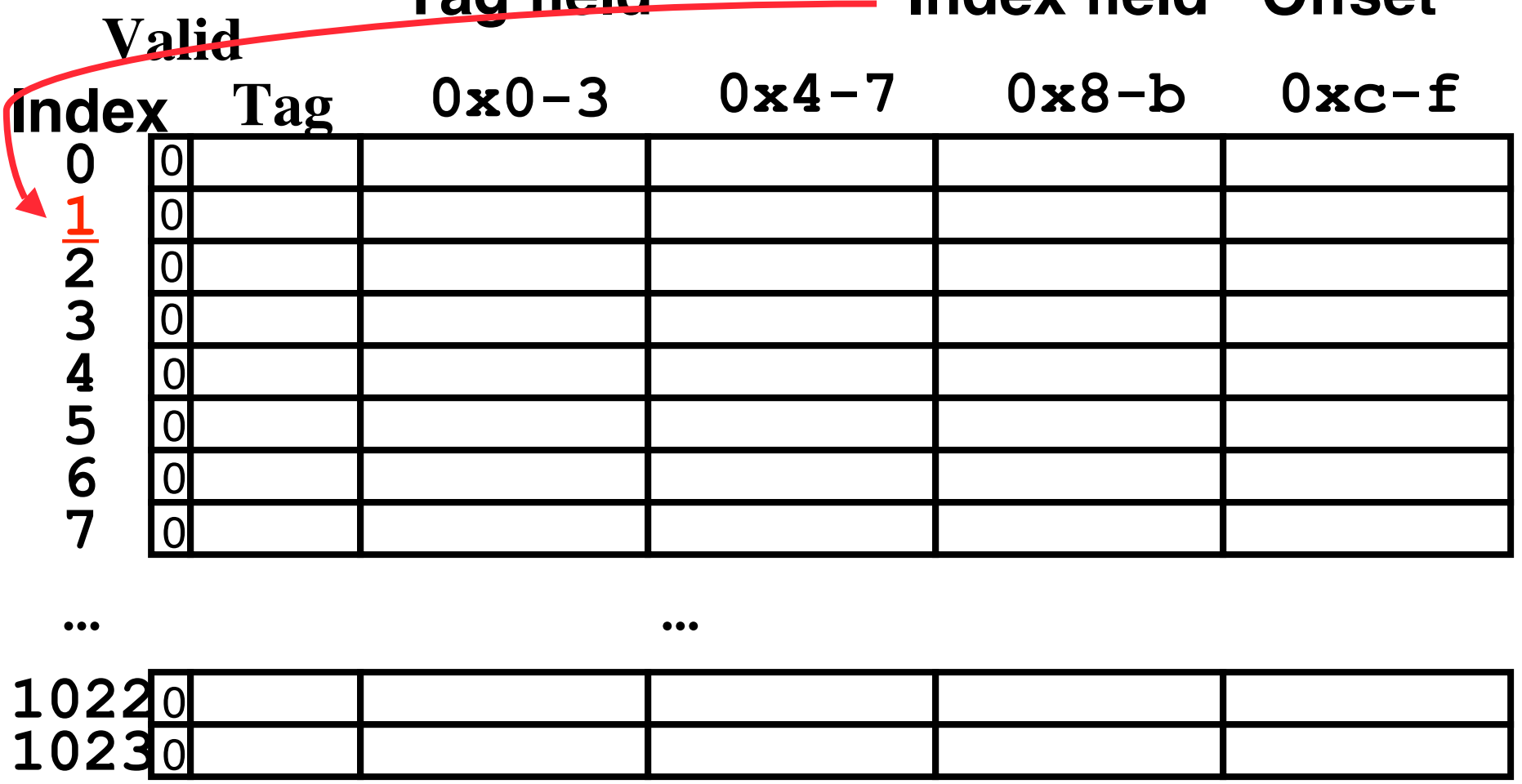
Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…			…			
1022	0					
1023	0					

So we read block 1 (0000000001)

- 00000000000000000000 0000000001 0100

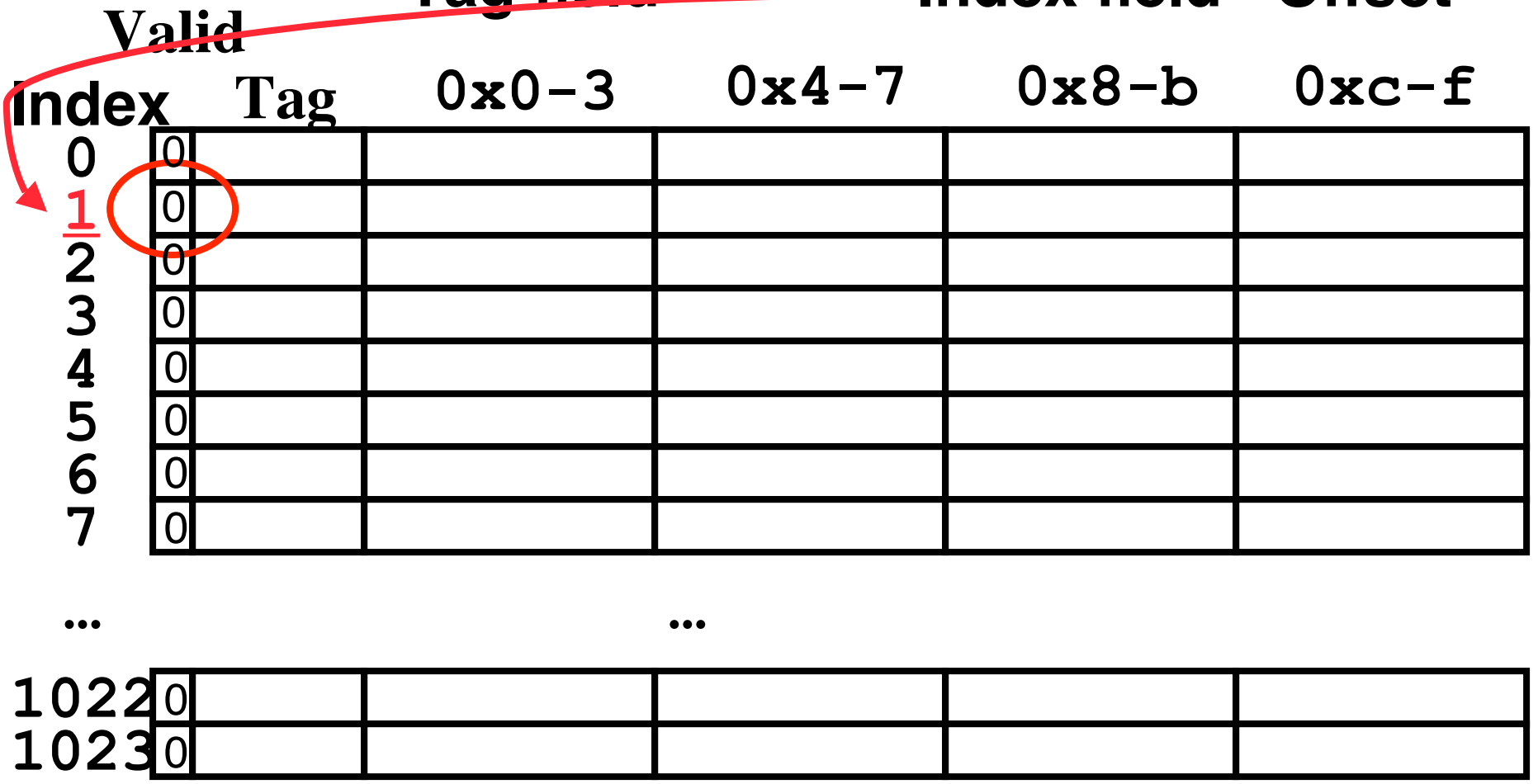
Tag field | Index field | Offset



No valid data

- 00000000000000000000 0000000001 0100

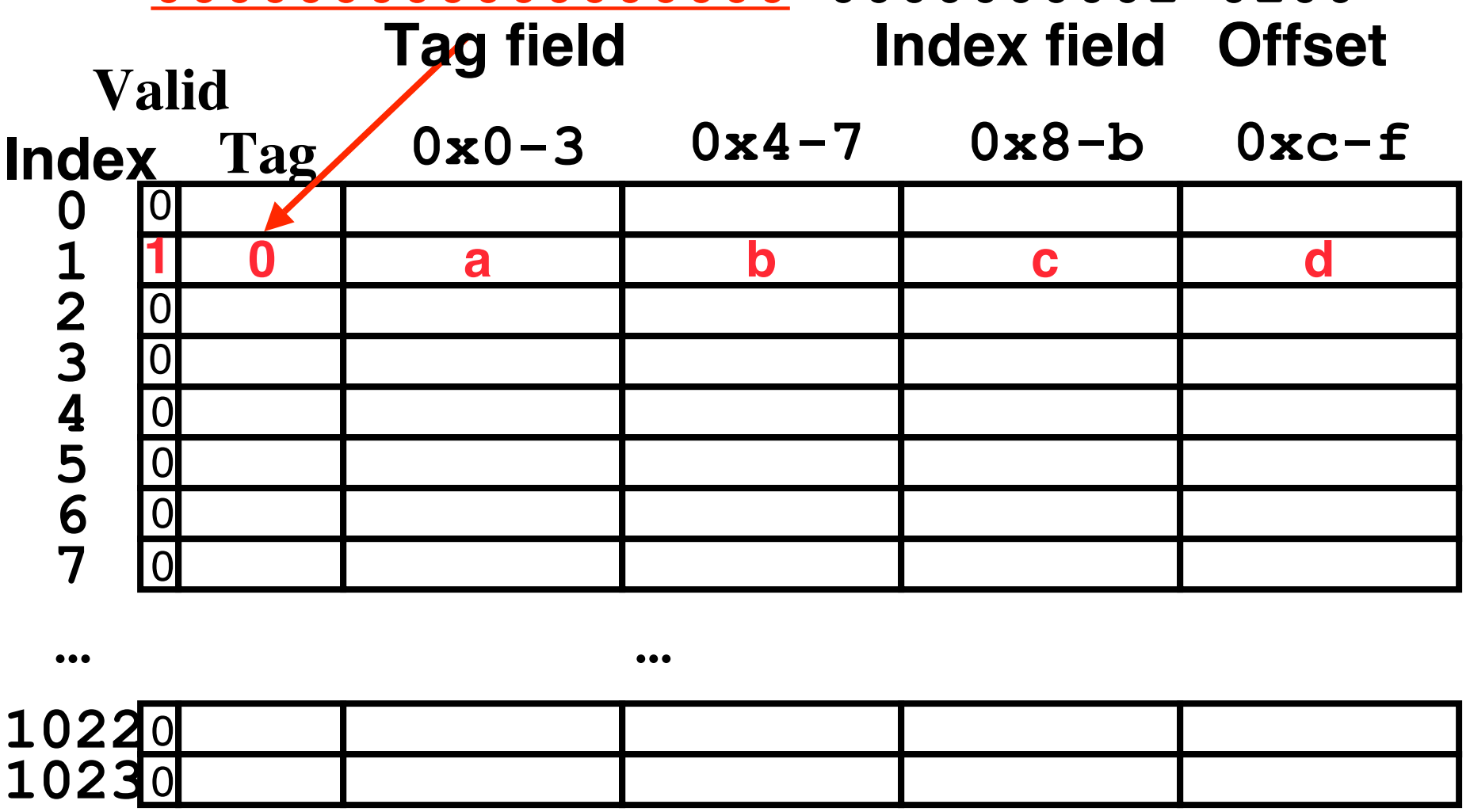
Tag field | Index field | Offset



So load that data into cache, setting tag, valid

- 00000000000000000000 0000000001 0100

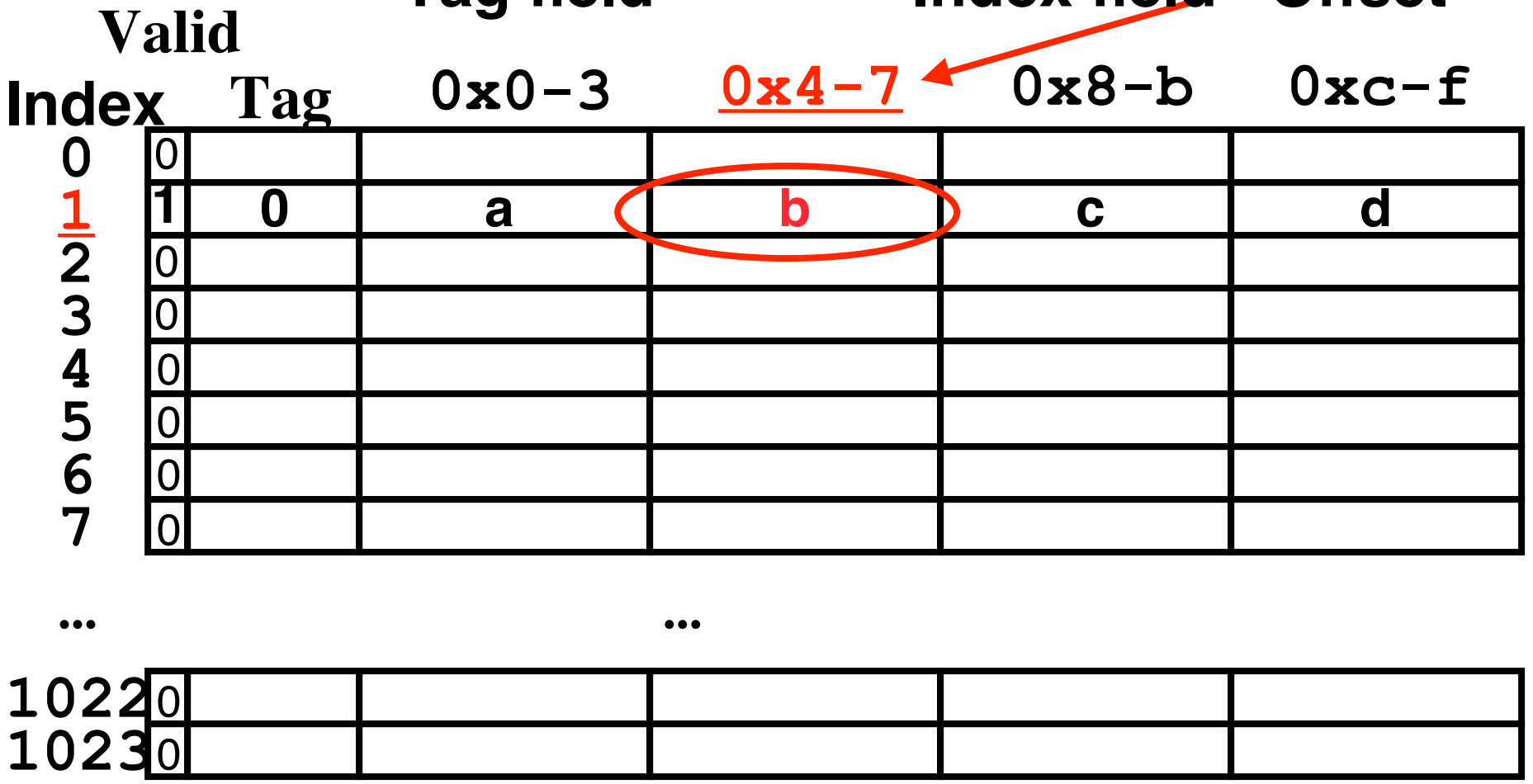
Tag field | Index field | Offset



Read from cache at offset, return word b

- 00000000000000000000 0000000001 0100

Tag field | Index field | Offset



2. Read 0x0000001C = 0…00 0..001 1100

- 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

Index is Valid

- 00000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Index valid, Tag Matches

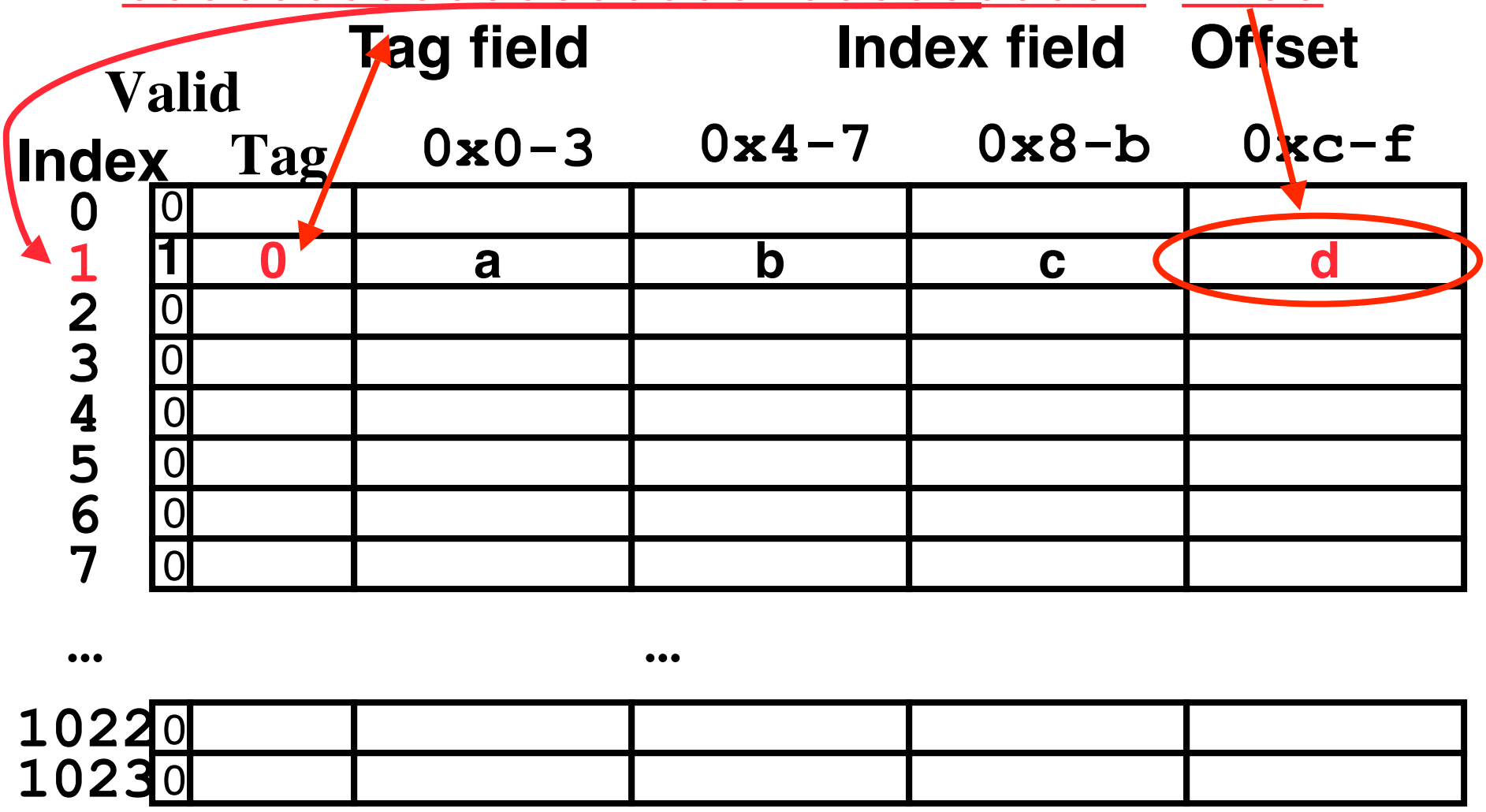
- 00000000000000000000 0000000001 1100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Index Valid, Tag Matches, return d

- 00000000000000000000 0000000001 1100



3. Read 0x00000034 = 0…00 0..011 0100

- 00000000000000000000 0000000011 0100

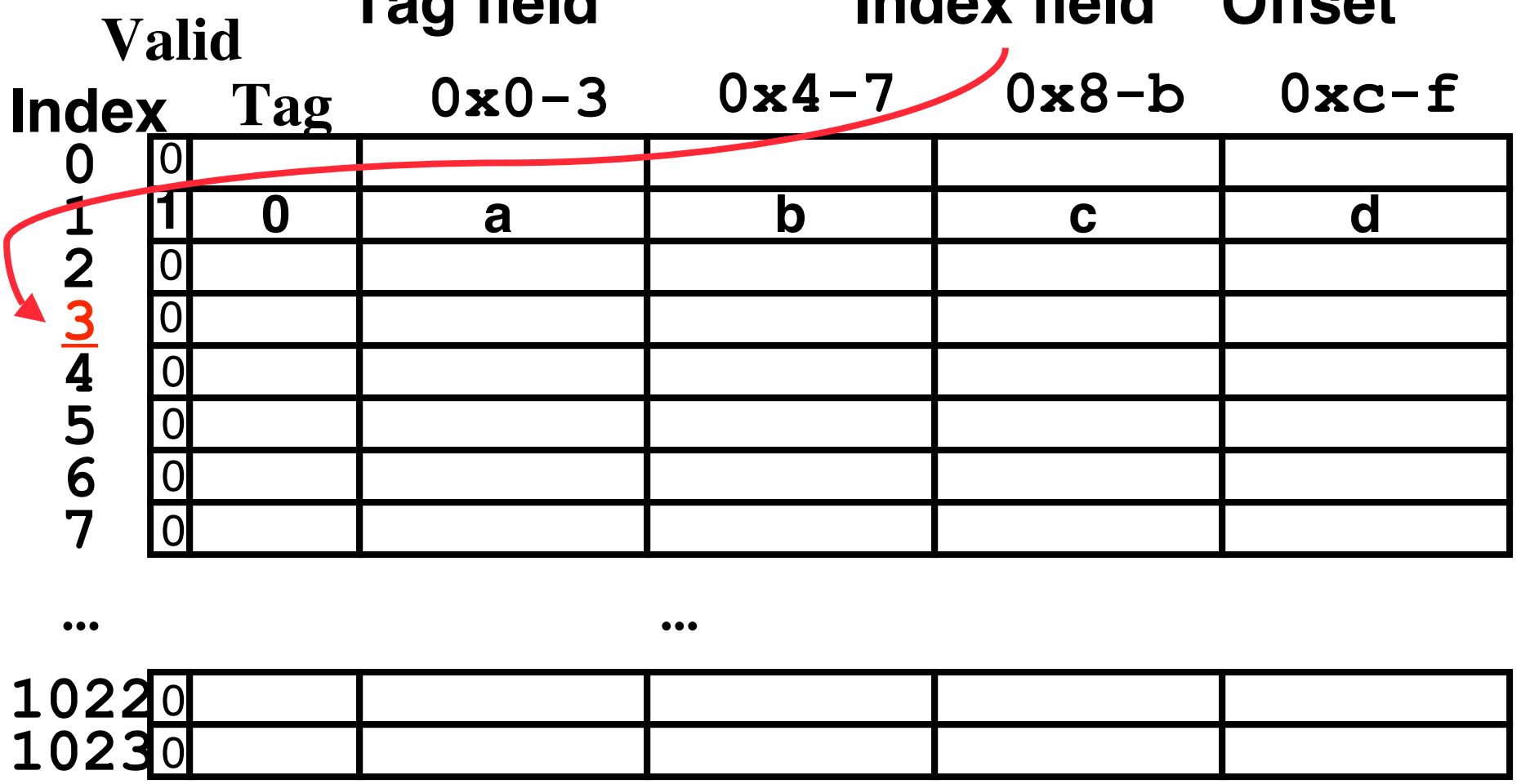
Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

So read block 3

- 00000000000000000000 0000000011 0100

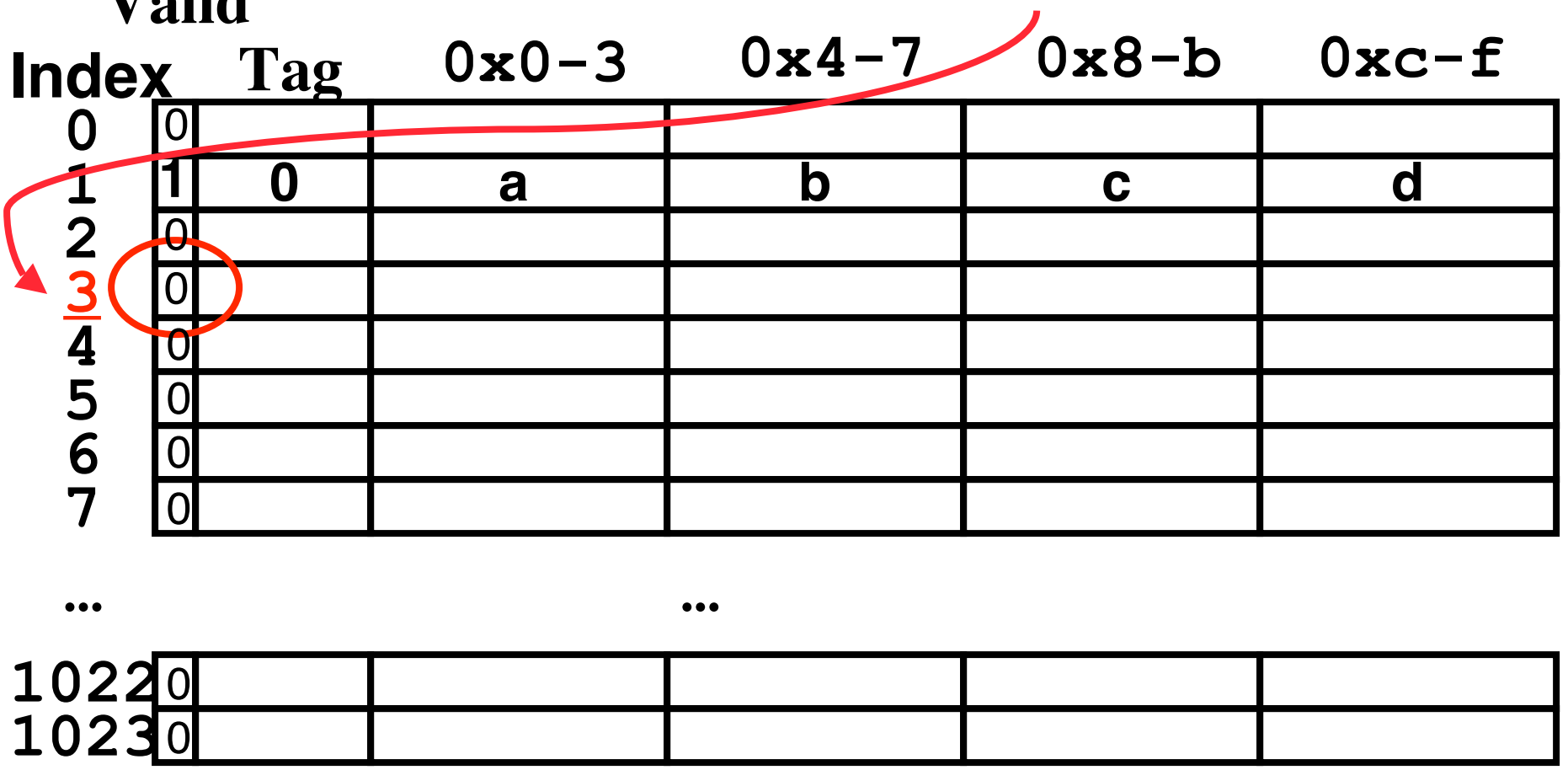
Tag field | Index field | Offset



No valid data

- 00000000000000000000 0000000011 0100

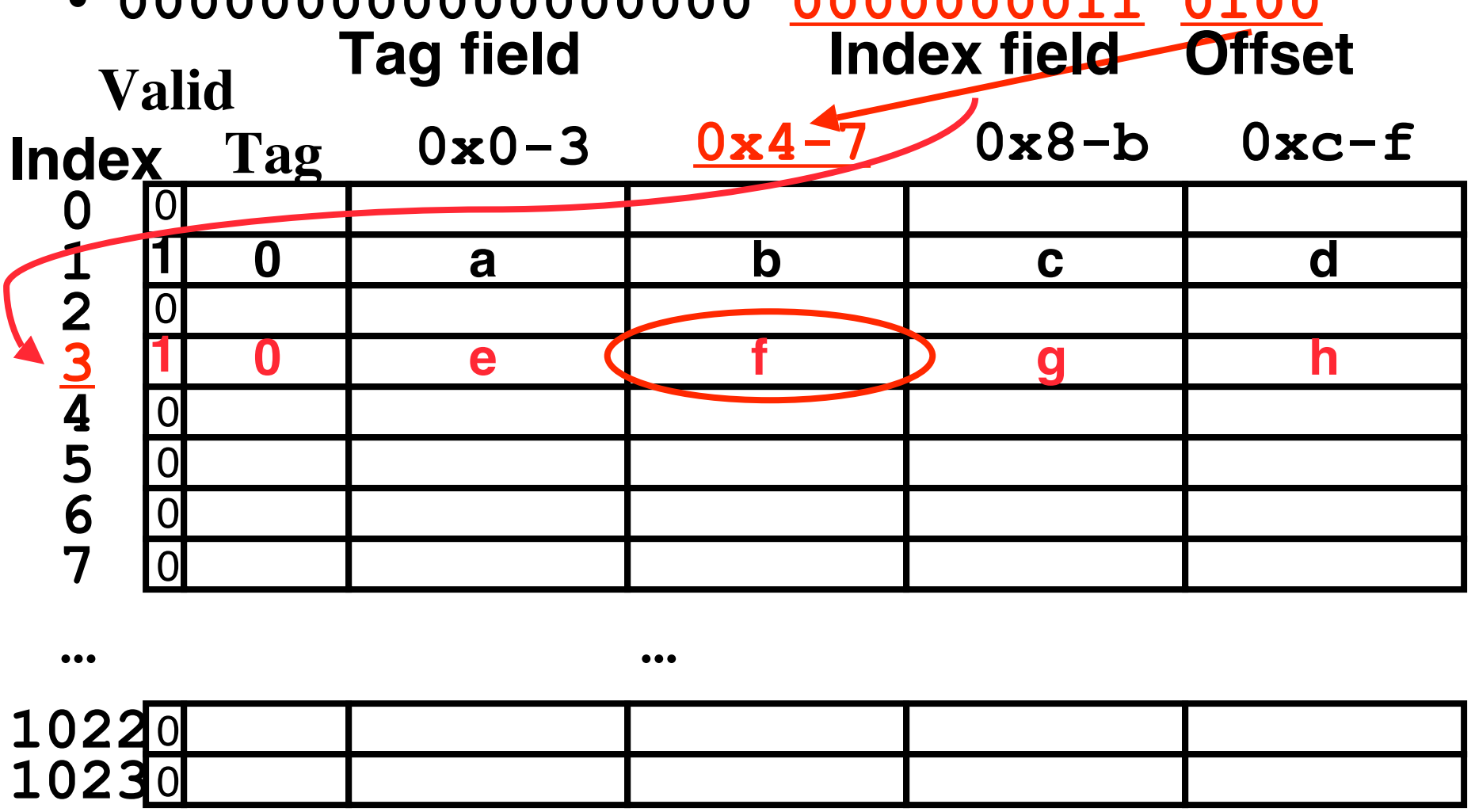
Tag field | Index field | Offset



Load that cache block, return word f

- 00000000000000000000 0000000011 0100

Tag field | Index field | Offset



4. Read 0x00008014 = 0…10 0..001 0100

- 00000000000000010 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
…			…			
1022	0					
1023	0					

So read Cache Block 1, Data is Valid

- 00000000000000010 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Cache Block 1 Tag does not match (0 != 2)

- 0000000000000010 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Miss, so replace block 1 with new data & tag

- 0000000000000010 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

And return word j

- 0000000000000010 0000000001 0100

Tag field | Index field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...			...			
1022	0					
1023	0					

Do an example yourself. What happens?

- Chose from: Cache: Hit, Miss, Miss w. replace
  Values returned: a ,b, c, d, e, ..., k, l
- Read address 0x00000030 ?
  `0000000000000000 0000000011 0000`
- Read address 0x0000001c ?
  `0000000000000000 0000000001 1100`

Cache

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					
...				...		

Answers

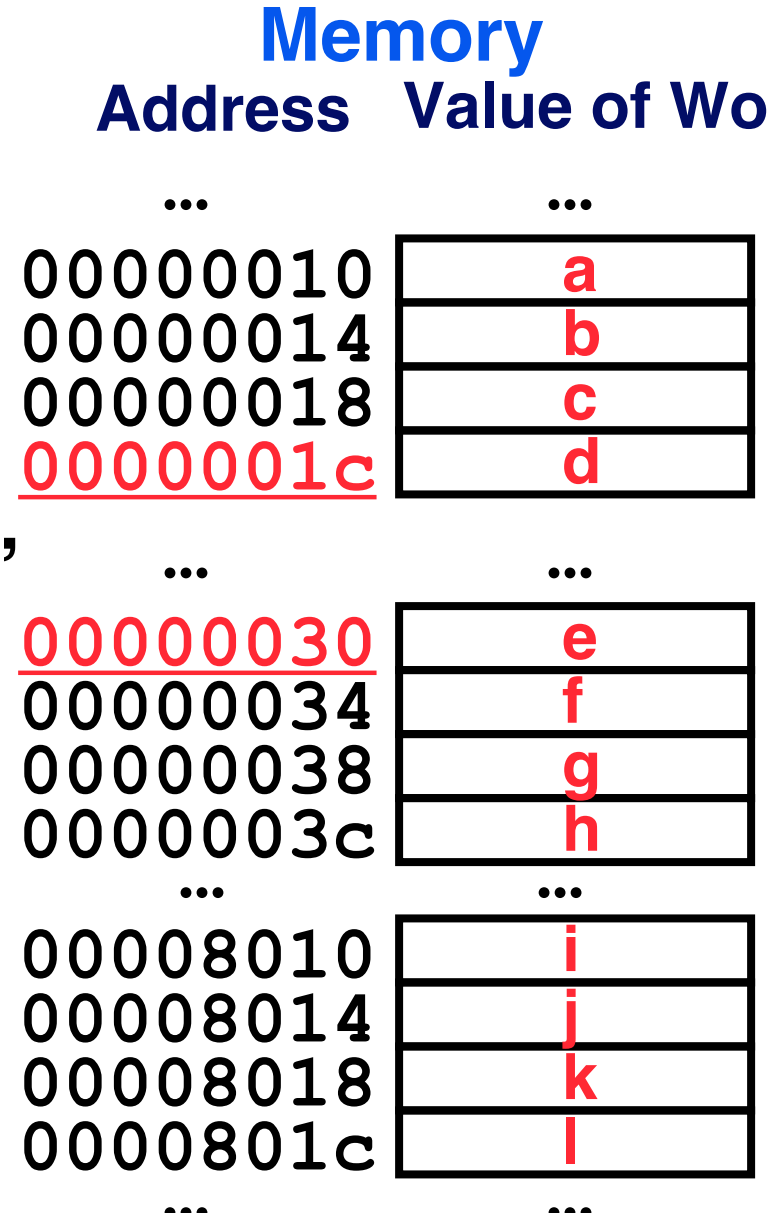
- 0x00000030 a hit

  Index = 3, Tag matches,
  Offset = 0, value = e

- 0x0000001c a miss

  Index = 1, Tag mismatch,
  so replace from memory,
  Offset = 0xc, value = d

- Since reads, values must = memory values whether or not cached:
  - 0x00000030 = e
  - 0x0000001c = d



Peer Instruction

A. Mem hierarchies were invented before 1950. (UNIVAC I wasn't delivered 'til 1951)

B. If you know your computer's cache size, you can often make your code run faster.

C. Memory hierarchies take advantage of spatial locality by keeping the most recent data items closer to the processor.

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Peer Instruction Answer

A. "We are…forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less accessible." – von Neumann, 1946

B. Certainly! That's call "tuning"

C. "Most Recent" items ⇒ Temporal locality

A. Mem hierarchies were invented before 1950. (UNIVAC I wasn't delivered 'til 1951)

B. If you know your computer's cache size, you can often make your code run faster.

C. Memory hierarchies take advantage of spatial locality by keeping the most recent data items closer to the processor.

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Peer Instructions

1. All caches take advantage of spatial locality.
2. All caches take advantage of temporal locality.
3. On a read, the return value will depend on what is in the cache.

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Peer Instruction Answer

1. All caches take advantage of spatial locality. **FALSE**
2. All caches take advantage of temporal locality. **TRUE**
3. On a read, the return value will depend on what is in the cache. **FALSE**

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

1. Block size = 1, no spatial!
2. That's the idea of caches; We'll need it again soon.
3. It better not! If it's there, use it. Oth, get from mem

And in Conclusion (1/2)

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible.
- So we create a memory hierarchy:
  - each successively lower level contains "most used" data from next higher level
  - exploits temporal locality
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- Locality of reference is a Big Idea

And in Conclusion (2/2)

- **Mechanism for transparent movement of data among levels of a storage hierarchy**
  - set of address/value bindings
  - address ⇒ index to set of candidates
  - compare desired address with tag
  - service hit or miss
    - load new block and binding on miss